Data Synchronisation in Parallel Unstructured Mesh Applications

Dries Kimpe[1,2], Stefan Vandewalle[1], and Stefaan Poedts[2]

[1] Technisch-Wetenschappelijk Rekenen, K.U.Leuven,
Celestijnenlaan 200A, 3001 Leuven, België
Dries.Kimpe@cs.kuleuven.be
[2] Centrum voor Plasma-Astrofysica, K.U.Leuven,
Celestijnenlaan 200B, 3001 Leuven, België

Abstract. Parallel computing is commonly used in computational fluid dynamics. For distributed memory machines – currently the most used architecture for large parallel machines – mesh partitioning is utilized to distribute the workload. In iterative codes, the cost of synchronising the partition borders will be the dominant factor in determining the parallel performance. As such, optimal synchronisation is of upmost concern.
MPI, the de facto programming model for these machines, traditionally offered multiple ways to implement this synchronisation. However, in practice, inefficiencies in the MPI libraries usually limited portable applications to a single method.
Advances in MPI implementations, interconnect networks and processor technology now changed the situation. In this work, we reevaluate boundary synchronisation. The communication pattern arising from the partitioning of different 2D and 3D meshes is studied, and used as input to a number of MPI synchronisation methods, both on distributed memory and on shared memory machines.

1 Motivation and Problem Description

Computational fluid dynamics (CFD) deals with the solution of a system of partial differential equations describing the motion of a fluid. This is commonly done by discretizing these equations on a mesh. Depending on the numerical algorithm, a set of unknowns is associated with either nodes or cells of the mesh. The amount of computational work is proportional to the number of cells. For realistic problems this quickly leads to simulations larger than a single system can handle.

Using mesh partitioning to speed up these simulations is by now an established practice. However, something recently changed: the ever increasing grow of per-core processor performance has now stopped[1], and there is a clear evolution into the direction of increasing the number of cores instead. Some architectures[4, 5] take this one step further and trade per-core performance for a significant increase in the number of cores.

The overhead of the parallel simulation (mainly communication and synchronisation) will vary greatly with the mesh size and computational cost of each

cell. But one thing will remain true: when the same mesh is partitioned into more pieces, the synchronisation and communication cost will account for a growing part in the total solution time. The advent of the multi-core revolution thus only increases the importance of scalability since the same problems will now need to be partitioned to run on more cores.

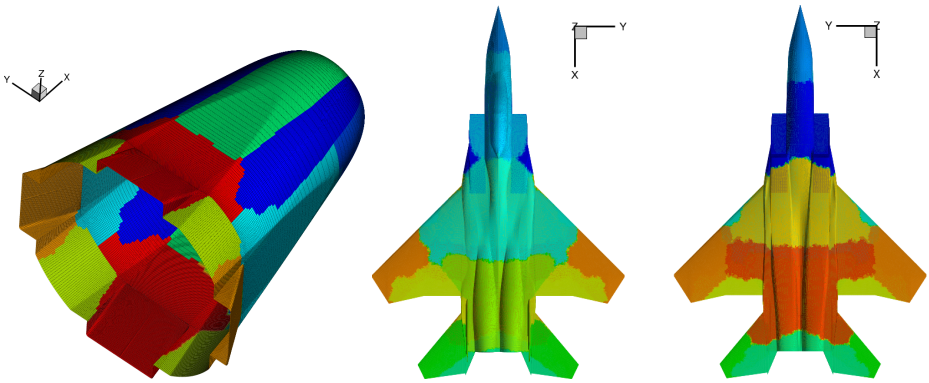
From the software point of view, things also changed. Changes in datatype processing, and the increased availability of MPI one-sided operations – together with the shared memory machines natively supporting them – calls for a reevaluation of long-standing synchronisation practices.

In this work, we investigate data synchronisation in parallel unstructured mesh simulations. Tests are performed on contemporary machines using modern MPI implementations. An attempt is made to define a best practice strategy.

2 Simulation Setup

2.1 Mesh Partitioning

For testing, a number of meshes from actual simulations were selected. Figure 1 shows some of the studied meshes. In (a), the surface mesh from a 3D mesh of approx. 3 million nodes is shown. This mesh was used in simulating the reentry of the EXPERT vehicle. For more information, see [2, 3]. The top and bottom surface mesh of an F15 jet figher is depicted in (b).



(a) EXPERT vehicle (2932000 nodes)

(b) F15 jet(621636 nodes)

Fig. 1. Example meshes (color indicates partition)

The way the mesh is partitioned directly affects the communication pattern. Not only amount of data that needs to be synchronised, but also the number of communication partners influences the performance of a given synchronisation method. Therefore, the logical communication pattern was also investigated.

Figure 2 shows an example of this, where the communication map of a small 2D simulation, running on 12 cores, can be seen. The local nature of the communication is clearly visible. In this example, each core has between 2 and 7 communication partners. The arrows indicate the direction and size of the data transfer, while the numbers between parentheses indicate the number of internal and boundary items.

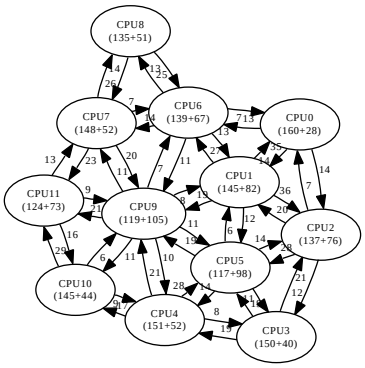


Fig. 2. Communication pattern in a small 2D simulation

2.2 Synthetic Load

We used a synthetic load generator, which busy waits a configurable amount of time per cell, as a replacement for the actual computation. This is done accurately capture compare the overlap potential of a given synchronisation method. On interconnect networks that support it, overlapping of communication and computation can have a major impact on the simulation time.

3 Synchronisation Methods

A wide range of synchronisation strategies were tested. They will be described in short below. For more details, we refer to the full paper.

Method 1: MPI Datatypes, non-blocking and persistent Method 1 is implemented in a way which offers the MPI implementations the most opportunity for optimization. There is a full datatype description, as well on the sender as on the receiver side. Additionally, all receives are non-blocking and posted before the sends are started. And finally, this method makes use of persistent send and receive operations.

In principle – given the right hardware – this means that the network hardware itself can scatter and gather data directly from memory (thanks to the use

of datatypes). Since the complete send and receive is described in advance, this method also offers an opportunity to use pipelined sending.

Method 2: Application buffer packing Method 2 uses dedicated send and receive buffers. On each rank, and for each communication partner, a sufficiently large send and receive buffer is created. At the start of the synchronisation phase, the memory buffer is filled, and as soon as a buffer is complete it is transmitted using a persistend send.

Thanks to the use of dedicated buffers, this method is able to restart the receives *at the end of the synchronisation operation*. This means that the process can receive data at any point during the calculation, even if it did not yet enter the synchronisation phase. As a consequence, this method is extremely tolerant for timing problems, since a process never has to delay a send operation because of lack of a remote receive buffer.

Method 3: One-side operations The MPI one-sided operations offer multiple ways of implementing this data exchange. On machines natively supporting active data transfers, the MPI library can take advantage of the shared-memory semantics of the one sided operations. Both active and passive operations were tested.

Method 4: MPI Collectives This method uses `MPI_Alltoallw`, the most general of the all-to-all communication functions. This method has several disadvantages and is provided only as a baseline comparison.

4 Conclusion

We investigated different border exchange methods in the context of parallel unstructured mesh applications. Using MPI as the communications library, these methods were evaluated using a range of actual CFD meshes. An attempt was made to describe a best practice for use in unstructured mesh simulations.

References

1. Sutter, H. 2005, "The free lunch is over: a fundamental turn toward concurrency in software", Dr. Dobb's Journal 30 (3)
2. Panesi, M., Lani A., Magin T., Pinna F., Chazot O. and Deconinck H., "Numerical investigation of the non equilibrium shock-layer around the EXPERT vehicle", AIAA 18th Computational Fluid Dynamics Conference, June 2007, Miami (US).
3. Lani A., Panesi M., Rambaut P., Kimpe D., Quintino T., Deconinck H. and Chazot O., "Simulation of thermal non equilibrium effects on the EXPERT vehicle", 2nd European Conference for Aerospace Sciences, July 2007, ISBN 978-2-930389-27-3
4. Gara, A., Blumrich, M. A., Chen, D., Chiu, G. L.-T., Coteus, P., Giampapa, M. E., et al. (2005). "Overview of the Blue Gene/L system architecture", IBM Journal of Reasearch and Development, 49, 195212.
5. M. Reilly, L. Stewart, J. Leonard, D. Gingold, "SiCortex Technical Summary", 2006.